ScaleXFS: Getting scalability of XFS back on the ring

Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won, *KAIST*

https://www.usenix.org/conference/fast22/presentation/kim-dohyun

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

ScaleXFS: Getting scalability of XFS back on the ring

Dohyun Kim Kwangwon Min Joontaek Oh Youjip Won
Department of Electrical Engineering, KAIST

Abstract

This paper addresses the scalability issue of XFS journaling in the manycore system. In this paper, we first identify two primary causes for XFS scalability failure: the contention between in-memory logging and on-disk logging and the contention among the multiple concurrent in-memory loggings. We then propose three key techniques to address the scalability issues of XFS; (i) Double Committed Item List, (ii) Per-core In-memory Logging, and (iii) Strided Space Counting. Contention between the in-memory logging and the on-disk logging is mitigated using the Double Committed Item List. Contention among the in-memory logging operations is addressed with Per-core In-memory Logging. Strided Space Counting addresses the contention on updating the global journaling state. We call the newly developed filesystem ScaleXFS. In modified `varmail` workload, the latency of metadata operation, `unlink()`, decreases to 1/6th from 0.59 ms to 0.09 ms under 112 threads against stock XFS. The throughput of ScaleXFS corresponds to 1.5×, 2.2×, and 2.1× of the throughput of the stock XFS under `varmail`, `dbench` and `mdtest`, respectively.

1 Introduction

Modern computing platforms are experiencing two technical advancements; storage devices are getting quicker and faster with sub-msec flush latency and the number of CPU cores in the commodity server is reaching the hundreds [16,29,38,39]. Considering these circumstances, modern filesystem designs face the new challenges, such as manycore scalability of the filesystem operation.

The filesystem is responsible for two types of operations; updating the state of the in-memory filesystem and synchronizing the in-memory filesystem state to the storage. A substantial effort has been dedicated to improving the throughput and the latency of synchronizing the filesystem state to the disk, also known as *filesystem journaling* [14, 35, 44, 47]. These works aim to hide flush latency: the latency of making the updated filesystem blocks durable in storage. A number of techniques have been proposed to make the flush latency invisible to the host, e.g. `no_barrier` mount operation [20], the adoption of the power loss protection feature at the flash storage [46], the kernel patch that omits the flush command [1], etc. Thanks to all these techniques, the latency to synchronize the filesystem to the storage has become less of a concern.

As the filesystem operation is relieved from the excessive flush latency, many works have focused on improving the throughput of filesystem journaling via increasing the concurrency in filesystem journaling. They include providing multiple running transactions [27, 40], providing multiple committing transaction [26,27,34,40,48], using the lock-free algorithm to manage the log block list in more concurrent manner [43].

In this work, we revisit the manycore scalability issue in the modern filesystem. In particular, we focus our effort on XFS. The XFS filesystem is one of the most widely used journaling filesystems in the enterprise server as well as in cloud platforms. XFS is the default filesystem for RHEL [2] which operates 33% of the enterprise servers [6] and handles more than 50% of stock transactions in the world [7]. Until recently, it has also been the default backend filesystem for distributed storage system, Ceph [13].

Despite the significance of XFS filesystems in modern computing platforms, few works have explored the performance and scalability aspect of the XFS and/or propose the solution in a rigorous manner. Existing articles on XFS are introductory article about the XFS data structures and algorithms [10], personal comment on the design of the XFS [19], comparison of the multiple filesystem performances without detailed analysis [12, 23, 37]. A few works coined performance and scalability issue of the XFS filesystem [13,37], whose solution has yet to be addressed.

Most works regarding filesystem scalability use EXT4 as their baseline filesystem [26, 27, 34, 40, 43, 48]. While both XFS [44] and EXT4 [35] use journaling to synchronize the filesystem state to the disk, few of the scalability techniques for EXT4 are readily applicable for XFS. This is because the details of their journaling subsystem designs lie at the other end of the extreme. XFS uses multiple granularity differential logging while EXT4 uses page granularity physical logging. XFS uses the copy of the update for journal commit while EXT4 uses the original page cache entry for journal commit. XFS allows multi-threaded concurrent journal commit while EXT4 has single threaded serial commit. As long as filesystem journaling is concerned, XFS adopts far more sophisticated data structure to make filesystem journaling more efficient and scalable. Most works that deal with EXT4 journaling scalability are about how to migrate the page cache entries among the running transactions (or committing transactions) when multiple threads update the same page cache

entry [24,26,27,34,40,48]. XFS is free from the page conflict since the filesystem operation creates its own copy of the update.

In this work, we examine the scalability of the XFS filesystem under three different workloads and perform in-depth analysis to find the prime cause for the observed scalability failure. Through this analysis, we find that the contention on the two locks are the root cause for its scalability failure and is caused by the contention among in-memory logging activities and the on-disk logging activities to access the global log data list which is called the *committed item list* in XFS. Furthermore, we find that the lock contentions observed in XFS are caused by two entirely different system behaviors.

The contribution of this work can be summarized as follows. First, we identify the root cause of the scalability failure in XFS; the contention on `cil_lock` and `ctx_lock-R` (shared lock of `ctx_lock`) that protect the committed item list. Second, we identify the essential filesystem activity that causes the contention on these locks: (i) the contention between in-memory logging and on-disk logging and (ii) the contention among the multiple concurrent in-memory loggings. Third, we propose ScaleXFS to address the scalability issue in XFS. We extend XFS with the techniques proposed below (Linux kernel v5.8.5). ScaleXFS consists of three key technical ingredients as follows.

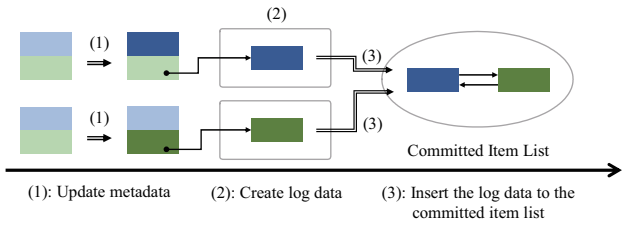
- **Double committed item list** XFS provides a single global committed item list. Unlike stock XFS, ScaleXFS provides two committed item lists so that in-memory logging and on-disk logging can work on its own committed item list. Double Committed item list prohibits the on-disk logging from blocking the in-memory logging activity.
- **Per-core in-memory logging** ScaleXFS forms each committed item list with a set of per-core committed item lists to mitigate the lock contention on the in-memory loggings. The application threads must acquire an exclusive lock on the committed item list when they insert the log data to the committed item list. Under the manycore system, multiple applications can perform in-memory logging concurrently and may compete for acquiring the exclusive lock on the committed item list. To mitigate the contention among the multiple concurrent in-memory loggings, ScaleXFS adopts a per-core committed-item list.
- **Strided Space Counting** XFS maintains a global state of the committed item list, which must be updated each time the log data is added to the committed item list. The application threads must acquire an exclusive lock on the global state of the committed item list when it updates the state. To mitigate the contention on accessing the global state of the list, we adopt sloppy counter [17] style per-core distributed counting scheme for maintaining the state of the committed item list. We call this, *Strided Space Counting*.

We believe that the beauty of ScaleXFS is its elegant simplicity. We overhaul the XFS journaling behavior, precisely identify the scalability bottleneck and propose a simple yet elegant mechanism that addresses the scalability issues in the XFS filesystem with minimal modifications. Unlike the other approaches that modify the on-disk layout of the existing filesystem partition [26, 27, 34, 40], the filesystem layout remains unchanged. ScaleXFS can mount the existing XFS partition. In ScaleXFS, the latency of `unlink()` decreases to 1/6th under modified `varmail` workload [1] [45], compared to stock XFS. In modified `varmail` and `dbench` [11] workloads, ScaleXFS renders as much as 1.5× and 2.2× performance against stock XFS, respectively. In ScaleXFS, the performance scales well till the number of cores reaches sixty while in stock XFS, the performance saturates beyond twelve cores under `mdtest` [33].

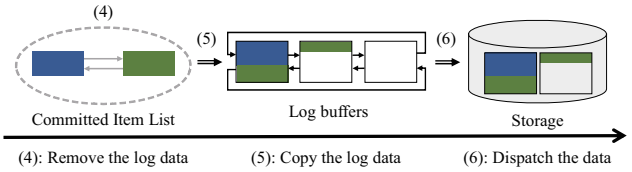
2 Background

2.1 XFS

XFS has been introduced to address the scalability issue of then Unix filesystem [44] (e.g. EFS [42] and FFS [36]). It supports a full 64-bit filesystem. It partitions a filesystem into a number of fixed-size partitions, the *allocation groups*, for scalable access to the filesystem metadata [44]. It uses B+ tree to manage the free blocks and the free inodes, respectively, and it uses hashing and B+ tree to manage a large number of directory entries within the directory.



(a) Filesystem Update



(b) Filesystem Journaling (journal thread)

Figure 1: Updating the filesystem object and logging the associated updates to the disk.

For journaling, XFS adopts multi-granularity differential logging [28, 30]. XFS applies different logging granularity subject to the size of the metadata that is updated. For small metadata such as inode, it creates a copy of the updated metadata for logging. For large metadata, e.g. superblock (4KByte)

[1] Each thread works on its own separate directory.

or B+ tree node (4KByte), it partitions the 4 KByte block into 128 Byte units and creates a copy of the updated region in 128 Byte granularity [9]. Each metadata object of XFS maintains the logs for the associated updates. The updates are synchronized to the disk either through periodic flush (30 sec by default) or by `fsync()` call.

For each metadata update, XFS creates the log data for journaling, which is a copy of the update. XFS maintains three types of log data lists: (i) log data that needs to be committed to the disk (*committed item list*), (ii) log data that is being committed (*committing list*) and (iii) log data that needs to be checkpointed (*active item list*). The filesystem thread creates and migrates the log data among these lists. The application threads and the journaling thread lock these lists and the associated data structure to avoid race condition. The contention among these threads leaves the XFS under potential scalability failure [44].

We can categorize the filesystem operations into two; metadata operation, e.g. `creat()` and `unlink()` and journaling operation `fsync()`. We explain the details of the metadata operation and the journaling operation in the following sections from the aspect of filesystem journaling.

2.2 Metadata Operation in XFS

For metadata operation, the filesystem acquires the exclusive lock on the metadata it needs to update. Then, it updates the metadata and performs *in-memory logging*. In-memory logging consists of two phases: (i) creating the log data and (ii) inserting the newly created log data into the committed item list. In the first phase, the application establishes the shared lock on the committed item list and creates the log data. When the application modifies the metadata for which log data is already in the list, it replaces the existing log data with a newly created one. In practice, a committed item list is a list of the keys (`Log Item`) each of which refers to the associated log data. Separating the key from the value, the log data can be replaced with a new one in the committed item list without deleting and inserting the entry in the committed item list. When the application updates the metadata whose log data is already in the committed item list, the associated entry in the committed item list is updated to refer to the newly created log. In the first phase of in-memory logging, the application establishes a shared lock on the committed item list. By using the shared lock, multiple applications (or threads) can concurrently update the log entries in the committed item list. The shared lock prohibits the journal thread from committing the committed item list if in-memory logging is in-progress. For the second phase, the filesystem establishes the exclusive lock (spinlock) on the committed item list and inserts the newly created log data. XFS inserts the new log data at the end of the committed item list. When it updates the existing log data, the updated log data is moved to the end of the committed item list.

Fig. 2 illustrates the case when the same inode is updated twice before the updates are committed to disk. First, the file attribute of the inode was updated to A'. The associated log is inserted to the committed item list. Second, the inline data of the inode is updated to B'. In the second update, the application creates the new log data which harbors both updates, [A',B'], and replaces the existing log [A'] with the new log data of [A',B'].

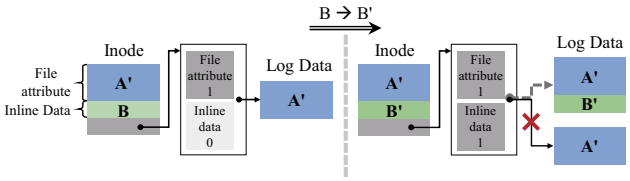


Figure 2: Differential Logging and the log management: An inode is updated twice. The first update is for the file attribute. The second update is for the inline data of the inode.

The filesystem maintains the state of the committed item list, e.g. the total size of the log data in the list. The filesystem updates this state when the log data is added to (or deleted from) the list. After the log data is inserted (or deleted) in the committed item list and the state is updated, the application releases the exclusive lock (spinlock) and the shared lock in the reverse order in which they are acquired.

2.3 Journaling Operation in XFS

Filesystem journaling is triggered through periodic journal flush (30 sec by default), by `fsync()` call, or when the size of the outstanding logs in memory exceeds a certain threshold. We define the filesystem journaling operation as on-disk logging for short. XFS defines *log buffer* as a unit of IO in on-disk logging. The default size of the log buffer is 32 KByte. XFS organizes the on-disk logging with two separate tasks and dedicates different threads for each. In this work, we call these two types of threads as the *journal thread* and the *commit thread*, respectively.

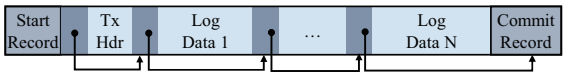


Figure 3: Structure of Transaction in Log Buffer.

XFS creates a new journal thread each time when the on-disk logging is triggered and can start new on-disk logging before the preceding one completes. The newly created journal thread establishes an exclusive lock on the committed item list and migrates the log data from the committed item list to the committing list. Then, it populates the log buffer with the log data in the committing list. If the log buffer becomes full, the journal thread flushes the log buffer to the disk (with `PREFLUSH` and `FUA` flags [22]). If all log data are copied to

the log buffer, the journal thread puts the commit record at the end of the sequence of the log data in the log buffer.

Fig. 3 illustrates the structure of the transaction in the log buffer. When the journal thread writes the commit record at the log buffer, the journal thread returns.

In copying the log data to the log buffer, XFS first moves the log data in the committed item list to the committing list and then releases the exclusive lock on the committed item list. After the journal thread migrates the log data from the committed item list to the committing list, the journal thread continues on-disk logging such as populating the log buffer with the log data from the committing list and flushing the log buffer when it is full. After the journal thread releases the exclusive lock on the committed item list, the committed item list becomes available for subsequent in-memory logging or on-disk logging. XFS allocates the non-overlapping partition of the log buffer to each journal thread. A number of journal threads can concurrently update the non-overlapping partitions of the same log buffer.

The committed item list has a unique transaction ID. It is allocated each time when new on-disk logging starts. When the journal thread holds the exclusive lock on the committed item list, it assigns the transaction ID to the committed item list. The filesystem monotonically increases the transaction id each time when the new journal thread is created, i.e. when the new on-disk logging starts. XFS maintains a set of transaction IDs that are being committed. When the journal thread finishes migrating the log data in the committed item list to the log buffer, it inserts the associated transaction id to the set of committing transactions. When the transaction is made durable, the transaction ID is removed from this set. All log data in the same committed item list are committed to storage as a single transaction.

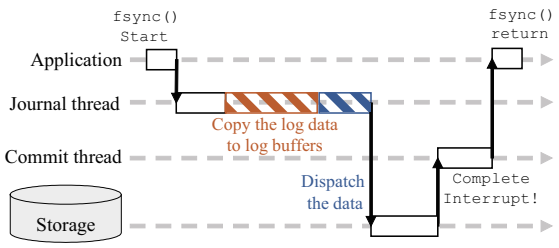


Figure 4: `fsync()` in XFS.

The commit thread is responsible for finishing the journal commit. When the host receives an interrupt informing that the log buffer has been made durable, the host creates the commit thread to handle the rest of the journal commit procedure. The commit thread marks the status of the log buffer as durable (`done_sync`). It then checks if all preceding log buffers have been made durable. If any of the preceding log buffers have not been made durable yet, the commit thread returns without finishing the on-disk logging. If it ensures that all preceding log buffers have been made durable, it releases the log buffer, removes the transaction ID from the set of IDs of outstanding transactions, and migrates the log data in the transaction to the checkpoint data list (`Active Item List`). If there exist the following log buffers that have been made durable and which have been waiting for its preceding log buffer to become durable, the commit thread also performs the same task for the associated log buffer, e.g. releases the log buffer and migrates the committed log data to the active item list. Then, the commit thread returns, and the caller of `fsync()` unblocks. Fig. 4 illustrates the execution of `fsync()` in XFS.

2.4 Concurrency in XFS journaling

XFS adopts a sophisticated mechanism for the filesystem scalability. The first is differential logging. Differential logging not only saves log space but also eliminates the conflict between the metadata operation and the journaling operation. Each metadata operation creates its own version of the update and is free from conflict with the other in-memory logging operations [9]. Also, the metadata operation and the journaling operation can proceed in parallel without interfering with each other. In EXT4, the application is blocked when it attempts to modify the page cache entry that is being committed to the journal region. The second is concurrent journal commit; XFS allows multiple transaction commits in flight. The third mechanism is out-of-order on-disk logging; XFS allows multiple threads to commit the journal transaction concurrently and independently. Journal thread places `cycle number` at each disk block of the journal region. The cycle number represents the number of times a given disk block is overwritten. By placing the cycle number at the beginning of each disk block, the journal threads can write the log blocks to the storage in an out-of-order manner and yet the recovery module can recover the filesystem to the correct state in case of the system failure.

3 XFS Scalability

3.1 Scalability Analysis

We conducted a experiment to examine the scaling behavior of the XFS filesystem. In our experiment, two different storage devices and three different workloads were used. Two storage devices, one with and one without the Power-Loss Protection feature, were used: Intel Optane 905P (I905) and the Samsung 970Pro (S970). The three workloads correspond to varmail per-thread DIR (W_{vptd}) which is a variant of the varmail workload of filebench (`varmail-ptd`) [45], client workload of `dbench` (W_d) [11], and `mdtest` (W_m) [33]. Each of the three workloads exhibit different frequency of metadata operation (`creat()` and `unlink()`) and the different intensity of the filesystem journaling (`fsync()`). We develop

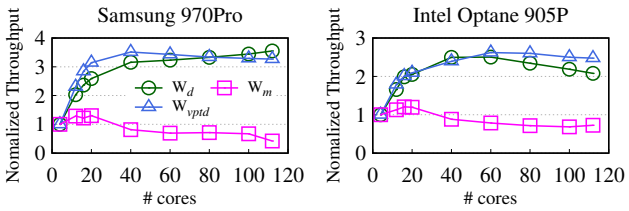


Figure 5: Performance Scalability of XFS. Throughput is normalized against the throughput with the four cores. Server: HPE ProLiant DL580 Gen10 with 112 cores, 4 socket (28 cores per socket), 512 GB DRAM.

the `varmail-ptd` to eliminate the contention on the shared directory in the original `varmail` workload. In the original `varmail`, multiple threads work on a shared directory for creating and deleting the files. The contention on the shared directory occurs at the VFS layer and may make the scaling behavior of the underlying filesystem invisible from the outside. The `varmail-ptd` allocates the separate directory for each thread. The `varmail-ptd` is the most journaling-intensive workload among the three workloads used here. In `varmail-ptd`, `fsync()` accounts for 15.4% of its system calls. In `dbench`, `fsync()` accounts for only 1% of the system calls. There is no `fsync()` in `mdtest`. Table 1 summarizes the characteristics of the three workloads.

Workload	`creat()`	`unlink()`	`rename()`	`read()`	`write()`	`fsync()`
`varmail-ptd`	7.7 %	7.7 %	0 %	15.4 %	15.4 %	15.4 %
`dbench`	17.3 %	3.5 %	0.7 %	27.2 %	8.6 %	1.2 %
`mdtest`	50 %	0 %	0 %	0 %	50 %	0 %

Table 1: Characteristics of workloads: Ratio of individual file system operations.

We use a 112 core machine (HPE ProLiant DL580 Gen10, 28 cores per socket, four sockets) to run all three workloads. We vary the number of active cores from four to 112. The number of threads were set to be equal to the number of cores, and the workload throughput was normalized against the throughput of four active cores. The results are illustrated in Fig. 5. All these workloads fail to scale. The performance of `mdtest` saturates with sixteen cores, and the performance of `dbench` and `varmail-ptd` saturates beyond thirty cores.

To identify the source of scalability failure, we measure the latency of the individual system calls in these workloads, `creat()`, `unlink()` and `fsync()`. Fig. 6 illustrates the results. The `mdtest` is omitted since it does not call `fsync()`.

When `fsync()` becomes quicker, e.g. due to the adoption of the high performance storage device, SSD with Power-Loss-Protection, or the filesystem mounted with `no-barrier` option, the latency of metadata operation, e.g. `unlink` becomes more dominant compared against `fsync()` latency. In `varmail-ptd` (Fig. 6a), the latency of metadata operation increases as `fsync()` gets quicker. This is due to the increased

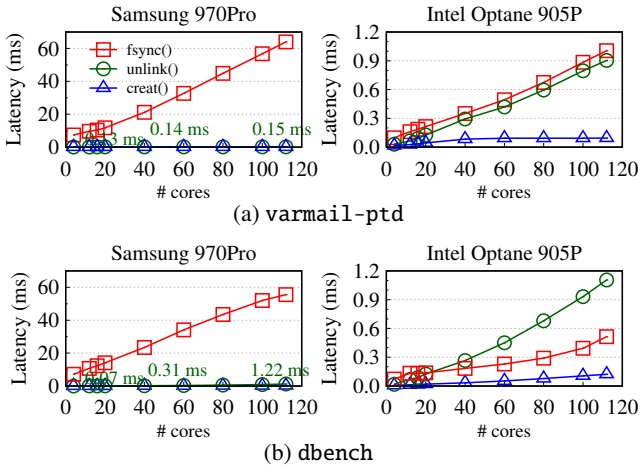


Figure 6: Average latencies of `creat()`, `unlink()` and `fsync()`.

contention on the shared object in the metadata operation. For `varmail-ptd` (Fig. 6a), the `fsync()` latency decreases from 64 ms to 1 ms and the `unlink()` latency increases by 6× from 0.15 ms to 0.904 ms when we compare the performance of S970 and I905. For `dbench` (Fig. 6b), the `fsync()` latency decreases from 56 ms (S970) to 0.51 ms (I905). In I905, due to shorter `fsync()` latency, the `unlink()` latency is 2.1× of the `fsync()` latency.

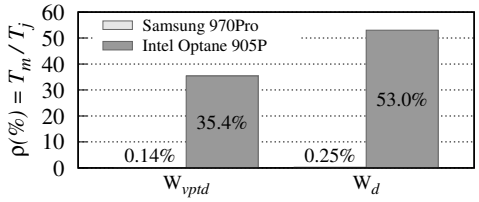


Figure 7: Percentage of lock wait time to T_j. (T_m = Wait time for acquiring lock in in-memory logging, T_j = Wait time for journaling IO in `fsync()`.

For finer analysis, we examine the wait time for acquiring the lock (lock wait time) in in-memory logging and the wait time for journaling IO in `fsync()`. XFS uses two locks; read-write semaphore (`ctx_lock`) and spinlock (`cil_lock`) to protect the committed item list and the log data from the race condition. The lock wait time for in-memory logging denotes the wait time for acquiring these two locks (`ctx_lock` and `cil_lock`). The wait time for journaling I/O in `fsync()` denotes the latency to writing the log blocks to the storage. This corresponds to the length of time interval from when the journal thread puts the commit record at the log buffer till when `fsync()` returns. Fig. 7 illustrates the ratio between the lock wait time and the wait time for journaling IO in S970 and I905, respectively. In S970, the lock wait time is less than 1% of the latency of journal I/O under both workloads. In I905,

the lock wait time becomes more significant. The ratio of lock wait time to wait time for journaling IO is 35.4% and 53.0% under `varmail-ptd` and `dbench`, respectively, in I905.

3.2 Component Analysis

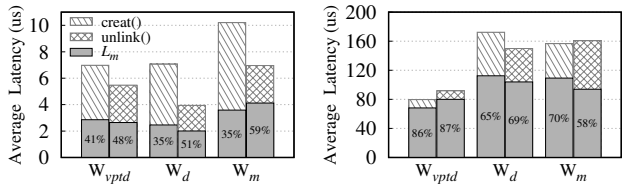


Figure 8: Average latency of `creat()` and `unlink()` and the ratio of the in-memory logging, L_m: the latency of in-memory logging. Storage Device: I905.

Based on the aforementioned experimental results, we found that the metadata operations, `creat()` and `unlink()`, are the prime suspect for the scalability failures in all three workloads. We examine the details of these filesystem operations and identify the main cause of scalability failure. The metadata operation consists of the metadata update and in-memory logging. We run three different metadata intensive workloads; `varmail-ptd`, `dbench` and `mdtest`. In each workload, we measure the latency of the metadata update and the latency of in-memory logging operations for `creat()` and `unlink()`, respectively. Fig. 8 illustrates the overhead of in-memory logging in `creat()` and `unlink()`, respectively. To examine performance behavior while varying the number of cores, we compared the latencies of metadata operations under four cores and 112 cores. In both workloads, the latency of the in-memory logging increases substantially with the increase in the number of cores. When there are four cores, the time for in-memory logging accounts for less than 50% of the latency of metadata operation in most cases (Fig. 8a). When there are 112 cores, in-memory logging accounts for as much as 90% of the latency of metadata operation in W_{vptd} workload (Fig. 8b).

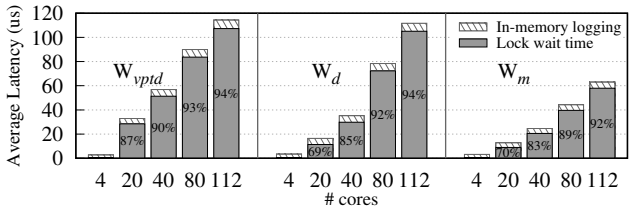


Figure 9: Lock wait time in in-memory logging, Storage Device: I905.

We examine the details of the in-memory logging overhead with the varying number of cores. We measure the wait time for acquiring the lock (lock wait time) and the actual time to update the committed item list and the log data. As in Fig. 9, the lock wait time increases with the number of cores while the actual time for inserting the log data to the committed item list remains unchanged regardless of the number of cores. When there are 112 cores, the lock wait time accounts for more than 90% of the total in-memory logging time.

3.3 Lock Contention Analysis

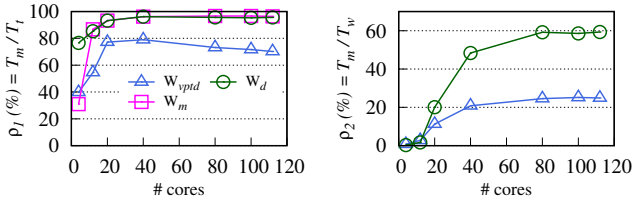


Figure 10: Percentages of lock overhead for in-memory logging to (a) or (b). T_m = Wait time for in-memory logging, T_t = Wait time for all locks, T_w = Workload Execution time. Storage Device: I905.

To precisely identify the scalability failure cause in in-memory logging, we analyze the lock contention in the filesystem. We use `Lockstat` [5] to obtain the lock wait times which are shown in Fig. 10.

Fig. 10a shows the ratio between the lock wait time associated with the in-memory logging (`ctx_lock` and `cil_lock`) against the total lock wait time in the filesystem under three workloads. The total wait time in the file system includes the wait time for all locks. The lock wait time associated with the in-memory logging accounts for a dominant fraction of the total wait time in `dbench` and `mdtest`, more than 90% beyond 20 cores. The lock contention overhead in the in-memory logging is far more severe than the lock contention on the filesystem metadata.

Fig. 10b shows the result of the ratio of the lock wait time for in-memory logging examined against the total workload execution time. When there is a small number of cores, e.g. ten cores, the lock wait time for in-memory logging accounts for less than 5 % of the total execution time. When there are 80 cores, the lock wait time accounts for 60% of the total execution time in `dbench`. In `varmail-ptd` with 80 cores, 25% of the total execution time is spent on the lock wait time for in-memory logging.

Through this in-depth analysis, we found that the lock contention associated with `ctx_lock` and `cil_lock` is the main cause of scalability failure in the three workloads.

4 ScaleXFS

4.1 Design Overview

XFS employs sophisticated techniques to reduce the time for committing a journal transaction to the storage. This is to hide the slow flush latency of the storage device. The techniques include differential logging, concurrent journal commit, out-of-order on-disk logging, re-logging [9], etc. These works are for reducing the amount of logs written to the journal region or for eliminating the ordering dependency within and between the journal transactions. On the other hand, the in-memory logging aspect of the XFS has not received proper attention. The advancement of the low latency storage devices [4] combined with the introduction of the manycore machine that has hundreds of CPU cores introduces another dimension of the complexity in the XFS journaling design. Due to the introduction of low latency storage devices loaded with power loss protection, the importance of efficiently handling the journal commit is less emphasized. Contrarily, as we observed so far, in-memory logging has become the major bottleneck in the performance and scalability of the XFS filesystem. In this work, we focus on resolving the scalability issue in the in-memory logging of XFS.

We observe that the lock that are used to protect the global objects in in-memory logging are the main cause of the scalability failure. There are two main objects in filesystem journaling in XFS: the committed item list and log data. These are used by both in-memory logging as well as on-disk logging.

The objective of this work is to mitigate the contention on the committed item list. In XFS journaling, there can be three types of contention on the committed item list; between in-memory logging and on-disk logging, among the concurrent in-memory loggings, and among the concurrent on-disk loggings. The first is between the application thread and the journal thread. The second is among the application threads that perform the metadata operation. The third is among the journal threads, which we find to be almost non-existent. We propose three key techniques to make the in-memory logging of XFS scalable; (i) To mitigate the contention between the in-memory logging and on-disk logging, we develop *Double Committed Item List*, (ii) To mitigate the contention among the in-memory loggings, we develop *Per-core In-memory Logging* and (iii) To mitigate the contention on the global state of the committed item list, we propose *Strided Space Counting*.

4.2 Double Committed Item List

We propose *Double Committed Item List* to eliminate the contention between the on-disk logging of the journal thread and the in-memory logging of the application thread. In the Double Committed Item List, we define two committed item lists. When the application thread finds that one of the committed item lists is being exclusively used by the journal thread, the application thread inserts the log item to the other committed item list. In ScaleXFS, a metadata object maintains up to two versions of the updates, one for each committed item list. The application may attempt to modify the metadata whose log data is being committed to the disk. If this happens, ScaleXFS inserts the newly created log data to the other committed item list than the one that is subject to (and locked by) on-disk logging. With a dual committed item list, in-memory logging and on-disk logging can proceed in parallel each of which works on its own committed item list. Fig. 11 illustrates an example of this. Application threads, T_1, T_2 and T_3 are accessing the committed item list for in-memory logging. These threads hold the shared lock on this committed item list. Journal thread T_{jrnl} is accessing the other committed item list for on-disk logging. T_{jrnl} holds the exclusive lock on this committed item list.

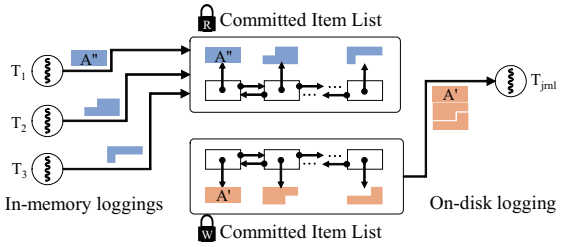


Figure 11: Double Committed Item List.

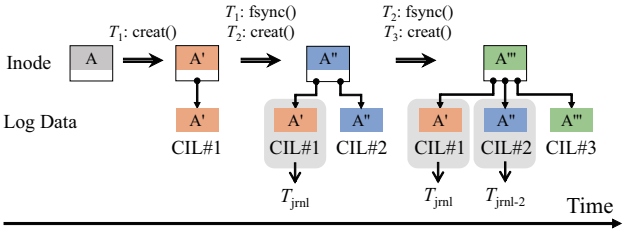


Figure 12: Multiple Committed Item Lists for in-memory logging: Triple Committed Item Lists.

Fig. 12 illustrates how the update log is inserted to the committed item list when we use three committed item lists. In practice, ScaleXFS is deliberately designed to use only *two* committed item lists in its design. In Fig. 12, there are three threads, T_1, T_2 and T_3. The three threads share a directory and update it, e.g. `creat()`. After they update the shared directory, each of them invokes `fsync()`. The first thread, T_1, updates the inode from A to A'. The log data is inserted to the committed item list 1. Then, T_1 calls `fsync()` to commit A' to the disk. While the log data A' is being committed, The second thread, T_2, updates the inode to A". Since committed item list 1 is being locked for on-disk logging, the application inserts the log data A" to the committed item list 2. Assume that T_2 calls `fsync()`. The newly arriving `fsync()` locks

the committed item list 2 that has log data A" and starts on-disk logging for the committed item list 2. Assume that third thread, T_3, attempts to update the same inode to A"'. Then, T_3 will select one of the committed item lists that are available for logging, either committed item list 1 or committed item list 3, and inserts the log for A"' to the selected committed item list. In Fig. 12, T_3 selects committed item list 3 and inserts A"' to selected one.

# cores	4	20	40	60	80	112
T_{create} (us)	42.5	42.5	68.5	91.5	122.4	192.7
T_{access} (us)	4.8	9.2	16.9	21.3	28.4	43.3
T_{create} / T_{access}	8.9×	4.6×	4.1×	4.3×	4.3×	4.4×

Table 2: Time to create journal thread (T_{create}) and for accessing the committed item list (T_{access}) under the `varmail-ptd`. HPE ProLiant DL580 Gen10, 4 socket (28 cores per socket) with 512 GB DRAM, Intel Optane 905P.

As we define the larger number of the committed item lists, the contention between the in-memory logging and the on-disk logging may be further reduced, but the overhead of maintaining the multiple lists increases. With detailed physical experiment and analysis, we find that the optimal number of the committed item lists in ScaleXFS is *two*. In Linux OS, creating a thread is a serial activity. Through physical experiment, we measure the time to create a thread (T_{create}) and the length of time interval during which the journal thread holds an exclusive lock on the committed item list (T_{access}). During T_{access}, the log data is migrated from committed item list to committing list. In on-disk logging, the time to create a journal thread is at least 4× longer than the time during which the journal thread holds the exclusive lock on the committed item list (Table 2). Technically, there can be multiple on-disk logging activities in flight in XFS. In reality, it is unlikely that two ore more journal threads compete for the exclusive lock on the committed item list. It is unnecessary to allocate more than one committed item list to mitigate the contention among the journal threads.

To prohibit the journal threads from locking both committed item lists (though it is unlikely to happen) and blocking of in-memory logging due to the lack of available committed item list, we allow at most one journal thread to lock the committed item list. In our physical experiment, we confirm that the newly arriving journal thread rarely observes any of the committed item lists being locked by on-disk logging activity.

The committed item list can be in one of the three states: Standby, Active and Blocked. In the Standby state, the committed item list is empty. In the Active state, the committed item list has the log data in it. The committed item list in the Active state can be free or locked by the shared lock for in-memory logging. The committed item list changes to the Blocked state if the journal thread requests for the exclusive lock on the committed item list. If the committed item list is available, the exclusive lock request is granted immediately. If the committed item list has been locked by the shared lock, the journaling thread waits until the shared lock is released. In both cases, the state of the committed item list changes to the Blocked state and the subsequent request for the shared lock is blocked.

ScaleXFS selects the committed item list for in-memory logging using a simple flip-flop based algorithm. The journal thread is assigned a transaction ID. The transaction ID monotonically increases with an increment of one each time the journal thread is created. The journal threads alternate between the two committed item lists for on-disk logging. The journal threads with an odd transaction ID will be using the same committed item list, e.g. CIL 1, while the journal threads with an even transaction ID use the opposite, e.g. CIL 2, for on-disk logging. Based upon the transaction ID of the current journaling thread, XFS selects the committed item list for in-memory logging.

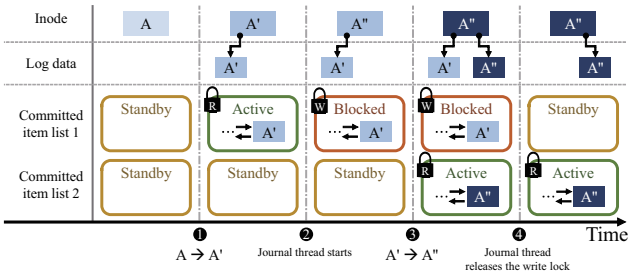


Figure 13: State of Committed item list.

Fig. 13 shows an example of the process of changing the state of the committed item list in double committed item list.

❶ **Standby** → **Active**: At the beginning, XFS creates the two committed item lists. Both of them are in the Standby state. Based upon the current transaction ID, we select the committed item list for in-memory logging and establish the shared lock. Once the shared lock is established, the state of the committed item list changes from Standby to Active state and the application thread performs in-memory logging on the selected committed item list. After the application thread finishes in-memory logging and releases the shared lock, the committed item list is still in the Active state. The subsequent application threads use the same committed item list since the current transaction ID has not changed yet.

❷ **Active** → **Blocked**: When the journal commit request arrives, the filesystem creates the journal thread increasing the transaction ID by one. Then, the journal thread establishes the exclusive lock on the committed item list and performs on-disk logging. The state of the committed item list becomes *Blocked*.

❸ **Standby** → **Active** If the application thread needs to perform in-memory logging while one of the committed item lists is in the Blocked state, the application thread chooses the other committed item list for in-memory logging. This committed item list goes into the Active state.

❹ **Blocked → Standby** The journal thread migrates all log data in the committed item list into the committing list. Once this is done, the committed item list becomes empty and the journal thread releases the lock on the committed item list. The committed item list goes into the Standby state.

We use `trylock()` [8] in establishing the shared lock on the committed item list. This is to avoid the situation where in-memory logging is blocked by the on-disk logging. If `trylock()` fails, ScaleXFS establishes the shared lock on another committed item list.

4.3 Per-core In-memory Logging

To mitigate the contention among the in-memory logging activities of the application threads, we propose *Per-core In-memory Logging*. The committed item list is organized with multiple lists on a per-core basis. With Per-core In-memory Logging, the application thread inserts the log data into the list allocated for the current CPU core on which it is being executed. Per-core in-memory logging mitigates the contention among the application threads that perform in-memory logging. In on-disk logging, the journal thread scans the per-core lists and merges them into a single list for on-disk logging. The rest of the on-disk logging procedure remains the same as when the committed item list is a single list of log data. Combined with the Double Committed Item List, each committed item list consists of a set of per-core lists (Fig. 14).

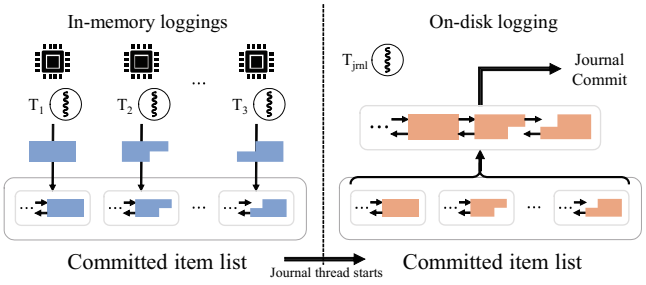


Figure 14: Per-core In-memory Logging.

There are two issues in organizing the committed item list with a set of per-core lists. First, we can preserve the order among the updates *only* within the per-core list. We cannot specify the ordering dependency among the logs in different per-core lists. Second, there can be a conflict among the per-core lists. Applications running on a different core may update the same metadata. There can be only one log data for each metadata object. The log data may need to be migrated between the per-core lists of different cores if the log data is updated by the threads running on the different cores.

We introduce *timestamp* for each log data to maintain the global order among the log data across the per-list lists. When the log data is newly created or updated, we put the timestamp

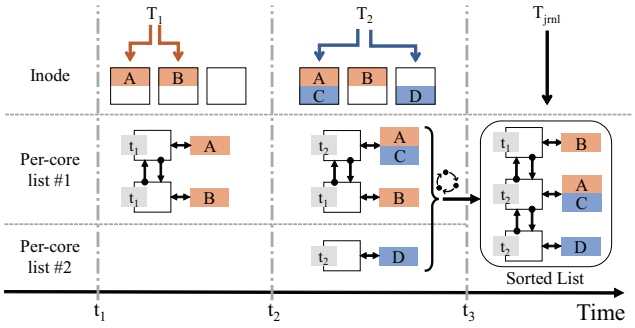


Figure 15: Ordering mechanism with timestamp: T_1 and T_2 modify the inodes at times t_1 and t_2, respectively.

at the associated log data, as illustrated by the example in Fig. 15. For the conflict, we insert the log data at the original per-core list where the preceding update exists.

At the beginning, there are two logs; A and B at per-core list #1. They are updated by thread T_1 at t_1. At time t_2, thread T_2 updates $inode_1$ to C and $inode_3$ to D. The per-core list #1 contains a log A for $inode_1$. When thread T_2 updates $inode_1$ to C, the log data in per-core list #1 is updated from A to A,C and the timestamp is also updated from t_1 to t_2. Thread T_2 creates the new log data for $inode_3$ and insert it to per-core list #2 with t_2. The updated log data remains at the original per-core list even though the thread at the different core updated it, e.g. at t_2 on per-core list #1 (Fig. 15). By pinning the updated log data to its original per-core list, we avoid modifying the per-core list across the core. In on-disk logging, the journal thread coalesces multiple per-core lists into a single list which maintains the global order of update with respect to the timestamp, e.g. at t_3 (Fig. 15).

4.4 Strided Space Counting

XFS maintains the total log data size in the committed item list. We call it a space counter. The space counter is used to estimate the size of disk space that is used to accommodate the logs in the committed item list. If the free space in the journal region is less than the space counter, XFS checkpoints the journal logs in the disk to make a room for the incoming logs. Each in-memory logging updates the space counter. The space counter is protected by spinlock (`cil_lock`).

To mitigate the contention on the space counter, we develop *Strided Space Counter*. Strided Space Counter is a variant of the sloppy counter [17] that is tailored to estimate the available space in the journal region of the disk. Strided space counter consists of the per-core space counters, the per-core strided space counter, the global space counter, and the stride length. The per-core space counter and the per-core strided space counter are initially set to 0. When the application performs in-memory logging, it increases the space counter by the size of the log data. If the local (per-core) space counter exceeds

the local strided space counter, the local strided space counter is folded to the global strided space counter. After it is folded to the global space counter, the local strided space counter increases by the length of stride. The thread increases the local counter when it needs to increase the space requirement. When the thread needs to check the space availability, it reads a global counter.

Strided space counter shares much of its behavior with sloppy counter but is not entirely the same. To avoid confusion, we call our counter Strided Space Counter. Sloppy counter estimates the *minimum* value of the sum of the local counters and is used to check if the reference counter value is non-zero [17]. Strided Space Counter estimates the *maximum* value of the sum of the local counters and is used to check if sufficient amount of space is reserved at the journal region. In sloppy counter, if the local counter exceeds the threshold value, the local counter is folded to the global counter. In Strided Space Counter, if updating the local counter makes the sum of the local counters greater than the global counter, we increase the value of the global counter by the stride length before we increase the local counter.

Time	Counter (L) / Strided Counter (S)								G
	L_1	S_1	L_2	S_2	L_3	S_3	L_4	S_4	
t_0	0	0	0	0	0	0	0	0	0
t_1	3	5	0	0	0	0	1	5	10
t_2	4	5	0	0	6	10	2	5	20
t_3	5	5	0	0	7	10	3	5	20
t_4	6	10	4	5	8	10	3	5	30

Figure 16: Stride Space Counting, `stride length`: 5.

Fig. 16 illustrates how strided space counting works. At t_1, the local counter at core 1 is changed to 3. The local strided counter of core 1 was 0. Now, it is updated to 5 by the stride length and it is folded to the global strided space counter. Simultaneously at t_1, the local counter at core 4 is change to 1. The local strided counter of core 4 is updated from 0 to 5 and it is folded to the global strided space counter. The global space counter G becomes 10 at time t_1. At time t_2, the local counter L_3 changes from 0 to 6. The local strided space counter becomes 10 and the global strided space counter becomes 20.

The stride length should be large enough to reduce the contention on the global counter. But, if the stride length is too large, the global space counter exceeds the required disk space for journaling too far and causes unnecessary checkpoint.

5 Evaluation

We examine the manycore scalability of ScaleXFS using a 112 core server (HPE ProLiant DL580, 28 core/socket, 4 sockets, Intel Xeon Platinum 8276) with 512 GByte DRAM. The CentOS 7.4 (kernel is 5.8.5) and Intel Optane 905p NVMe SSD were used. In the experiment, the number of active cores were varied and evenly distributed among the sockets. We compare three filesystems; EXT4, original XFS, and ScaleXFS. For ScaleXFS, we use three different settings; ScaleXFS$_{\mathrm{D}}$, ScaleXFS$_{\mathrm{DP}}$ and ScaleXFS$_{\mathrm{DPS}}$. Each subscript in ScaleXFS refers to a different feature. D, P and S refers to Double committed item list, Per-core in-memory logging, and Strided space counting, respectively.

5.1 Lock Contention

The goal of ScaleXFS is to eliminate the lock contention of shared journaling structures. We first show that ScaleXFS significantly reduces the lock contention of the overall system. `Lockstat` [5] was used to measure the lock contention to examine how it is mitigated in our work. For three workloads, the total lock wait time, the lock wait time for `cil_lock` and `ctx_lock-R` (shared lock of `ctx_lock`). were examined, and the results are shown in Fig. 17.

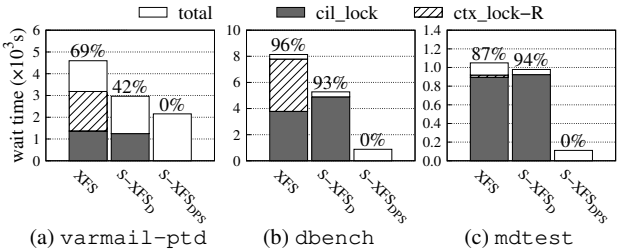


Figure 17: Total lock wait time of `cil_lock`, `ctx_lock-R` and all of the locks. Each number above each bar is the proportion of the sum of `cil_lock` and `ctx_lock-R` out of the total wait time of all locks, 112 cores.

Recall that `ctx_lock` protects the committed item list and the associated log data from the conflict between in-memory logging and on-disk logging. `cil_lock` protects them from conflicts among the multiple in-memory loggings. In XFS, `cil_lock` and `ctx_lock-R` combined account for as low as 69% (Fig. 17a), as high as 96% (Fig. 17b) of the total lock wait time. We do not observe any lock wait time for `ctx_lock-R` in `mdtest`. This is because `mdtest` does not issue any `fsync()` and therefore in-memory logging does not wait for the shared lock (`ctx_lock-R`).

It is shown that the double committed item list successfully eliminates the contention between in-memory logging and on-disk logging. ScaleXFS$_{\mathrm{D}}$ does not have any contention on `ctx_lock-R`, since in-memory logging and on-disk logging do not need to compete with each other anymore. By eliminating the overhead on `ctx_lock-R`, the total lock wait time also decreases by 36% and 35% on `varmail-ptd` and `dbench`, respectively. In ScaleXFS$_{\mathrm{DPS}}$, Per-core basis committed item list enables parallel in-memory logging, leading to the elimi-

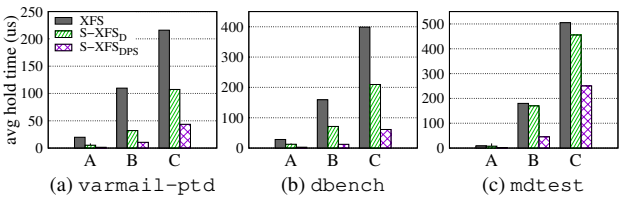


Figure 18: Average hold time of locks protecting metadata objects. A, B and C refer to `xfs_nondir_ilock_class-W`, `xfs_dir_ilock_class-W`, `i_mutex_dir_key` each. 112 cores.

nation of the contention on `cil_lock`. The total wait time of all locks decreases by 53%, 90%, and 90% for `varmail-ptd`, `dbench` and `mdtest`, respectively, in ScaleXFS$_{DPS}$ (Fig. 17).

As a result of reducing the in-memory logging latency, the ScaleXFS reduces the total time to hold the exclusive lock on the metadata object. This is because the filesystem needs to hold the exclusive lock on the metadata object until the metadata operation completes, i.e. until the in-memory logging completes. The lock wait times of three widely used locks, `xfs_nondir_ilock_class-W`, `xfs_dir_ilock_class-W` and `i_mutex_dir_key`, were examined under XFS, ScaleXFS$_D$ and ScaleXFS$_{DPS}$ filesystems. Three benchmark workloads, `varmail-ptd`, `dbench` and `mdtest`, were examined and their average hold times were measured at 112 cores. Fig. 18 illustrates the results. In `varmail-ptd`, ScaleXFS$_{DPS}$ reduces the hold time of each lock by 92%, 90%, and 80% against the original XFS. A similar improvement in the other workloads was also observed. Eliminating the lock contention associated with the in-memory logging and the on-disk logging, i.e. contention on `cil_lock, ctx_lock-R`, the entire lock hold time on the metadata update decreases to as much as 1/10th.

5.2 Latency of creat(), unlink() and fsync()

To show how the operation latencies are effectively reduced by eliminating lock contentions, we evaluated the average latencies of three filesystem operations; `creat()`, `unlink()`, and `fsync()` on `varmail-ptd` and `dbench` under a varying number of cores. In this experiment, we also include the performance result from EXT4.

ScaleXFS reduces the latency of metadata operations. The reduction becomes more pronounced as the number of cores increases. In `varmail-ptd`, `creat()` and `unlink()` latencies in ScaleXFS$_{DPS}$ are 1/5 and 1/6 of those in XFS at 112 cores, respectively (Fig. 19a and Fig. 19b). In `dbench`, `creat()` and `unlink()` latencies in ScaleXFS$_{DPS}$ are 1/4 and 1/6 of those in XFS, respectively (Fig. 19d and Fig. 19e). ScaleXFS$_{DPS}$ renders more scalable behavior than the XFS. Under `varmail-ptd` workload, when the number of cores increases from 4 to 112, in XFS and ScaleXFS$_{DPS}$, the latency of `unlink()` increases by 20× and by 3.5×, respectively.

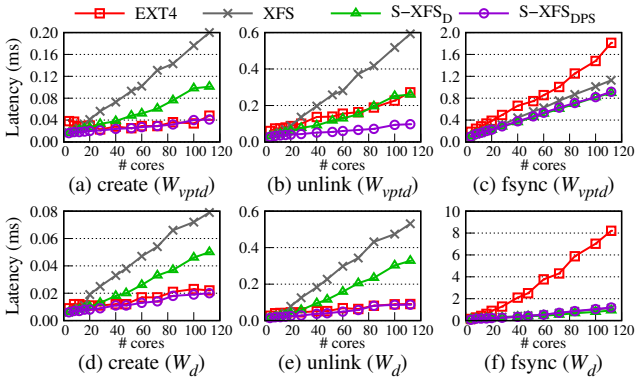


Figure 19: Average latencies of `creat()`, `unlink()`, `fsync()` at `varmail-ptd` (W_{vptd}) and `dbench` (W_d).

Fig. 19 shows that XFS and EXT4 have their own performance edges. XFS exhibits better performance in filesystem journaling than EXT4. EXT4 renders better performance in metadata operation than XFS. ScaleXFS$_{DPS}$ makes a significant improvement in the metadata operation against stock XFS. ScaleXFS$_{DPS}$ exhibits better filesystem journaling performance than XFS (Fig. 19c) and better performance in metadata operation than EXT4 (Fig. 19a and Fig. 19b).

5.3 Benchmark performance

`varmail-ptd`. Fig. 20a illustrates the result of `varmail-ptd`. ScaleXFS$_{DPS}$ outperforms XFS and EXT4 by 1.5×, 1.9× at 112 cores, respectively. ScaleXFS$_{DPS}$ scales well while XFS performance saturates beyond 20 cores. The performance difference between ScaleXFS$_D$ and ScaleXFS$_{DPS}$ is not significant in `varmail-ptd` because the on-disk logging accounts for a substantial fraction of the entire workloads, and subsequently contention among the in-memory loggings is not as severe as in the other workloads. Therefore, the benefit of using per-core in-memory logging is not significant.

`dbench`. Fig. 20b illustrates the result of `dbench`. ScaleXFS$_{DPS}$ shows 1.3× and 2.2× performance against ScaleXFS$_D$ and XFS, at 112 cores, respectively. ScaleXFS$_{DPS}$ also outperforms EXT4 by 4.5× at 112 cores. In `dbench`, per-core in-memory logging and Strided Space Counting becomes more effective than in `varmail-ptd`. In `dbench`, in-memory logging operation accounts for larger fraction of operation than in `varmail-ptd`. In `varmail-ptd`, on-disk logging (`fsync()`) accounts for 15.4% of the total number of system calls. In `dbench`, on-disk logging (`fsync()`) accounts for only 1.2% of the total number of system calls. (Table 1). In `dbench`, the contention among the in-memory logging operations becomes more intense than the contention between the in-memory logging and on-disk logging. The techniques to mitigate the contention among the in-memory loggings becomes far more effective in `dbench` than in

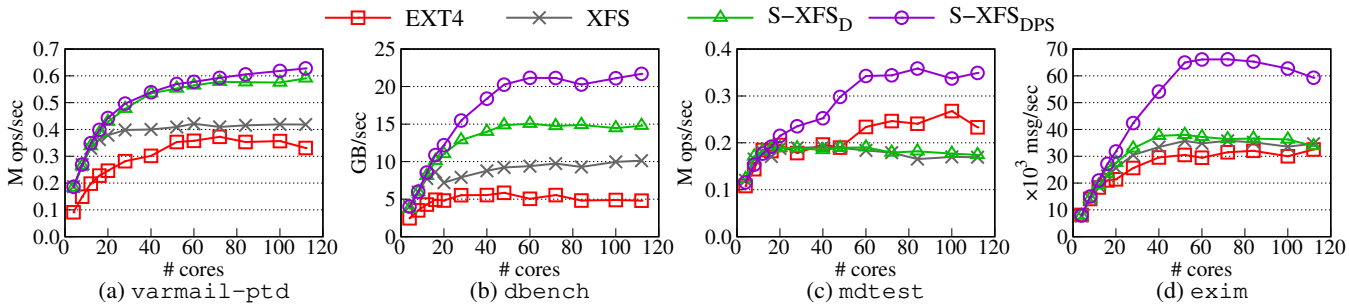


Figure 20: Throughput of `varmail-ptd`, `dbench`, `mdtest` and `exim`.

`varmail-ptd`. The performance gain of ScaleXFS$_{DPS}$ against ScaleXFS$_D$ becomes more substantial under `dbench` than under `varmail-ptd`.

The benefit of double committed item list is substantial in both `varmail-ptd` and `dbench` workload. ScaleXFS$_D$ renders 1.4× performance in `varmail-ptd` and 1.7× performance in `dbench` than XFS in 112 core server. In original XFS, on-disk logging (`fsync()`) blocks the in-memory logging. With double committed item list, in-memory logging and on-disk logging can proceed in parallel since each of them can work on its own committed item list. As a result, ScaleXFS$_D$ yields significant performance advantage against stock XFS in `dbench` and `varmail-ptd`.

`mdtest`. Fig. 20c illustrates the result of `mdtest`. ScaleXFS$_{DPS}$ outperforms the original XFS by 2.1× ScaleXFS$_{DPS}$ scales well till 60 cores while XFS saturates beyond 10 cores. ScaleXFS$_{DPS}$ adopts scalable in-memory logging and outperforms EXT4 by 1.5× `mdtest` is metadata intensive workload without `fsync()`. The performance benefit of adopting per-core in-memory logging is substantial in `mdtest`.

`exim`. Fig. 20c illustrates the result of `exim`. `Exim` [3] is a mail server, creating, renaming, and deleting small files repeatedly. We disable per-message `fsync()` in `exim` as in FxMark [37]. `Exim` does not issue `fsync()` similar to `mdtest` and the on-disk logging does not occur frequently. As a result, the effect of a double committed item list that is for removing the contention between the in-memory logging and the on-disk logging becomes less significant. The performance gap between ScaleXFS$_D$ and XFS is not remarkable. Contrarily, the contention among the in-memory loggings becomes more intense and the effect of per-core in-memory logging becomes significant. ScaleXFS$_{DPS}$ outperforms ScaleXFS$_D$ by 1.5×. ScaleXFS$_{DPS}$ outperforms XFS and EXT4 by as much as 1.9× and 2.2×, respectively. We observe that the throughput of ScaleXFS$_{DPS}$ slightly drops after 60 cores. This is caused by the overhead of spinlock in the VFS layer. The number of spool directories in `exim` is configured to 62. As ScaleXFS significantly increases the throughput, the overhead on spool directories becomes more substantial [31].

`FxMark`. Fig. 21 shows the result of `FxMark`. To analyze the performance improvement of the metadata operation, we examine the performance of `unlink()` under different sharing levels of FxMark [37].

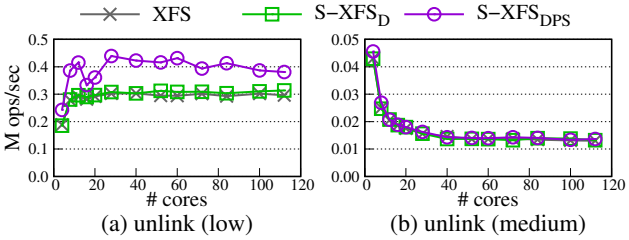


Figure 21: Throughput of FxMark's `unlink()` operation on low and medium sharing levels.

In the low sharing level, the processes perform the metadata operations in their private directory. ScaleXFS$_{DPS}$ exhibits superior performance against ScaleXFS$_D$ and XFS (Fig. 21a). Per-core in-memory logging is effective in eliminating the contention among the in-memory loggings.

In the medium sharing level, the processes perform the metadata operations in a shared directory. The contention to hold the exclusive lock on the shared metadata neutralizes the effectiveness of per-core in-memory logging, ScaleXFS$_{DPS}$ does not render any performance improvement against XFS (Fig. 21b).

5.4 Strided Space Counting

We measure the average latency and tail latency (@95%) of in-memory logging, and the throughput of `mdtest` under varying the stride lengths. We vary the stride length from 0 Byte to 8 KByte. When the stride length is 0 Byte, it represents the performance of ScaleXFS$_{DP}$. Stride space counting reduces the average latency as much as by 7.3% (Fig. 22a). The tail latency of in-memory logging is unaffected by strided space counting (Fig. 22b). Strided space counting improves the throughput by up to 13% compared against ScaleXFS$_{DP}$ (Fig. 22c). Dominant operations of `mdtest` workload is in-

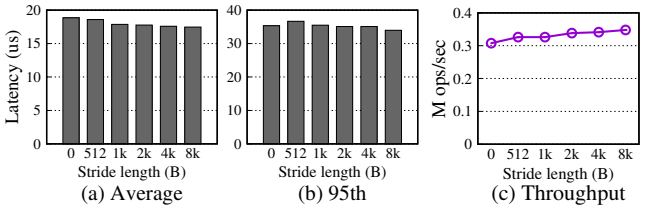


Figure 22: Effect of stride length: In-memory logging latency and throughput in `mdtest`, `creat()`, 112 cores.

memory logging. In manycore environment, large number of in-memory logging operations can proceed in parallel rendering intense contention on the shared variable such as locks and space counter. Via introducing the per-core space counter, ScaleXFS effectively mitigates the contention on the global variable, space counter substantially.

Larger stride length may render overestimating the size of the required journal region on the disk. We found that 8 KByte stride length yields 14 KByte of space waste for a 2 GByte journal region. We believe that space waste of 14 Kbyte for 8 KByte stride length is negligible. Given this, we set the default stride length to 8 KByte.

6 Related Work

A few works have examined the performance and scalability behavior of the XFS [13, 37]; however, they do not provide a detailed analysis on the observed behavior or offer a solution to address it. Other works explain the internals of XFS [10] and compare the performance of multiple filesystems [12, 23]. Few of these works perform an in-depth study on the performance bottleneck of the XFS filesystem and propose a solution to address the observed issues.

EXT4 suffers from sub-optimal performance primarily due to its page granularity physical logging that uses the original page cache entry for journal commit [48] and due to the serial journal commit [37]. A number of works proposed using multiple running transactions [27, 40] and/or multiple committing transactions [26, 27, 34, 40, 48] to address the performance and the scalability of the filesystem journaling. Son et al. [43] allows multiple processes to insert the log data to the running transaction concurrently via a lock-free mechanism. ScaleFS [15] decouples the in-memory filesystem from the on-disk filesystem. The in-memory filesystem adopts highly concurrent data structures with the per-core operation log and the on-disk filesystem merges the operation log and synchronize it to the disk.

Min et al. [37] found that the cache line bounce of the global lock causes a performance collapse in the manycore system. A number of works propose a scalable lock primitive. PRW lock [32] and RPS [31] adopt the per-core indicators to reduce the contention among readers. To improve the write lock latency, these works update the global value when the IPIs are transmitted [32] or when the per-core flags are set [31] by the writer. RPS efficiently checks the per-core indicators and flags, by leveraging the CPU scheduler [31]. BRAVO [21] uses the global hash table to reduce the memory footprint.

A number of works improve the scalability of the read operation. Refcache [18] adopts per-core reference delta caches and merges the changes between epoch into a single operation. PayGo [25] adopts the per-core hash technique and anchor counter for scalable counter. Lodic [41] achieves file block scalability by implementing a local counter on the page table entry considering the popularity of a file.

7 Conclusion and Future Work

In this work, we address the XFS filesystem scalability. We identify the prime cause for scalability failure in XFS journaling: the contention among the application threads and the journal thread to access the global list of log data. To address this issue, we propose Dual Committed Item List, Per-Core In-memory Logging and Strided Space Counting. Our experimental results confirm that ScaleXFS resolves the scalability bottleneck of XFS under various workloads. ScaleXFS reveals two new bottlenecks that were not visible before. The first one is the host-side overhead of handling on-disk logging. XFS allows that multiple committing transactions are made durable at the disk in out-of-order manner. However, to ensure the filesystem integrity, the XFS filesystem at the host-side finishes the journal commit in the order in which the transaction commit requests are made; XFS finishes the journal commit of a transaction only when all its preceding journal commit finishes. We find that this in-order completion for journal commit mechanism becomes a scalability bottleneck when we resolve the lock contention in in-memory logging. The second one is the memory copy overhead of differential logging. As the journaling becomes efficient, the memory copy overhead associated with creating the differential copy of the updated metadata accounts for a relatively larger fraction of journaling overhead. Currently, XFS adopts a crude coarse grain differential logging and it leaves substantial room for improvement. We like to address these two issues in our future effort.

8 Acknowledgements

We would like to thank our shepherd, George Amvrosiadis, for helping shape the final version of this paper. We are also grateful to the anonymous reviewers for their valuable feedback that have improved this paper. This work was in part supported by IITP of Korea (No. IITP-2018-0-00549), NRF of Korea (No. NRF-2020R1A2C3008525), and SNU-SK Hynix Solution Research Center (S3RC) (No. MOUS002S).

References

[1] http://lkml.iu.edu/hypermail/linux/kernel/1603.2/04523.html.

[2] Chapter 3. the xfs file system. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/ch-xfs.

[3] Exim. https://www.exim.org.

[4] Intel. breakthrough performance for demanding storage workloads. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-905p-product-brief.pdf.

[5] lockstat. https://www.kernel.org/doc/Documentation/locking/lockstat.txt.

[6] Red hat continues to lead the linux server market. https://www.redhat.com/en/blog/red-hat-continues-lead-linux-server-market.

[7] Red hat enterprise linux powers the globe's stock exchanges. https://www.redhat.com/en/blog/red-hat-enterprise-linux-powers-the-globes-stock-exchanges.

[8] trylock function. https://www.kernel.org/doc/htmldocs/kernel-locking/trylock-functions.html.

[9] Xfs delayed logging design. In *Filesystems in the Linux kernel*. https://www.kernel.org/doc/html/latest/filesystems/xfs-delayed-logging-design.html.

[10] SGI XFS Algorithms & Data Structures, 3rd Edition. 2006. http://ftp.ntu.edu.tw/linux/utils/fs/xfs/docs/xfs_filesystem_structure.pdf.

[11] Dbench. 2008. https://dbench.samba.org/.

[12] Mohd Bazli Ab Karim, Jing-Yuan Luke, Ming-Tat Wong, Pek-Yin Sian, and Ong Hong. Ext4, xfs, btrfs and zfs linux file systems on rados block devices (rbd): I/o performance, flexibility and ease of use comparisons. In *Proc. of ICOS*, pages 18–23. IEEE, 2016.

[13] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proc. of ACM SOSP*, pages 353–369, 2019.

[14] Steve Best. Jfs overview. how the journaled file system cuts system restart times to the quik. 2000. http://jfs.sourceforge.net/project/pub/jfs.pdf.

[15] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proc. of ACM SOSP*, pages 69–86, 2017.

[16] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proc. of the 44th annual design automation conference*, pages 746–749, 2007.

[17] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Tappan Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *Proc. of USENIX OSDI*, volume 10, pages 86–93, 2010.

[18] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proc. of ACM EUROSYS*, pages 211–224, 2013.

[19] J Corbet. Xfs: the filesystem of the future?, 2012.

[20] Jonathan Corbet. Barriers and journaling filesystems. https://lwn.net/Articles/283161/.

[21] Dave Dice and Alex Kogan. Bravo—biased locking for reader-writer locks. In *Proc. of USENIX ATC*, pages 315–328, 2019.

[22] C. HELLWIG. Patchwork block: update documentation for req_flush / req_fua. https://patchwork.kernel.org/patch/134161/.

[23] Christoph Hellwig. Xfs: the big storage file system for linux. *; login:: the magazine of USENIX & SAGE*, 34(5):10–18, 2009.

[24] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. Txfs: Leveraging file-system crash consistency to provide acid transactions. *ACM TOS*, 15(2):1–20, 2019.

[25] Seokyong Jung, Jongbin Kim, Minsoo Ryu, Sooyong Kang, and Hyungsoo Jung. Pay migration tax to homeland: anchor-based scalable reference counting for multicores. In *Proc. of USENIX FAST*, pages 79–91, 2019.

[26] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. Spanfs: A scalable file system on fast storage devices. In *Proc. of USENIX ATC*, pages 249–261, 2015.

[27] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. Z-journal: Scalable per-core journaling. In *Proc. of USENIX ATC*, pages 893–906, 2021.

[28] Yi-Reun Kim, Kyu-Young Whang, and Il-Yeol Song. Page-differential logging: an efficient and dbms-independent approach for storing data into flash memory. In *Proc. of ACM SIGMOD*, pages 363–374, 2010.

[29] George Kurian, Jason E Miller, James Psota, Jonathan Eastep, Jifeng Liu, Jurgen Michel, Lionel C Kimerling, and Anant Agarwal. Atac: A 1000-core cache-coherent processor with on-chip optical network. In *Proc. of 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 477–488. IEEE, 2010.

[30] Juchang Lee, Kihong Kim, and Sang Kyun Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Proc. of ICDE*, pages 173–182. IEEE, 2001.

[31] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A multicore-accelerated file system for flash storage. In *Proc. of USENIX ATC*, pages 877–891, 2021.

[32] Ran Liu, Heng Zhang, and Haibo Chen. Scalable read-mostly synchronization using passive reader-writer locks. In *Proc. of USENIX ATC*, pages 219–230, 2014.

[33] LLNL. mdtest. `https://sourceforge.net/projects/mdtest/`.

[34] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proc. of USENIX OSDI*, pages 81–96, 2014.

[35] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proc. of Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[36] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *Proc. of ACM TOCS*, 2(3):181–197, 1984.

[37] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proc. of USENIX ATC*, pages 71–85, 2016.

[38] Gibson Ming-Dar. Introduction of power loss protection function on ssd. 2019. `https://www.embeddedcomputing.com/technology/storage/introduction-of-power-loss-protection-function-on-ssd`.

[39] Timothy Prickett Morgan. Flashtec nvram does 15 million iops at sub-microsecond latency. 2014. `https://www.enterpriseai.news/2014/08/06/flashtec-nvram-15-million-iops-sub-microsecond-latency/`.

[40] Daejun Park and Dongkun Shin. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proc. of USENIX ATC)*, pages 787–798, 2017.

[41] Jeoungahn Park, Taeho Hwang, Jongmoo Choi, Changwoo Min, and Youjip Won. Lodic: Logical distributed counting for scalable file access. In *Proc. of USENIX ATC*, pages 907–921, 2021.

[42] Silicon Graphics, Inc. IRIX Advanced Site and Server Administration Guide, Chapter 8. `https://irix7.com/techpubs/007-0603-100.pdf`.

[43] Yongseok Son, Sunggon Kim, Heon Y Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *Proc. of USENIX FAST*, pages 227–240, 2018.

[44] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proc. of USENIX ATC*, volume 15, 1996.

[45] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[46] Kingston Technology. A closer look at ssd power loss protection. 2019. `https://www.kingston.com/en/solutions/servers-data-centers/ssd-power-loss-protection`.

[47] Stephen C Tweedie et al. Journaling the linux ext2fs filesystem. In *Proc. of Annual Linux Expo*. Durham, North Carolina, 1998.

[48] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled io stack for flash storage. In *Proc. of USENIX FAST*, pages 211–226, 2018.